iJRASET
International Journal For Research in Applied Science and Engineering Technology

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

**Volume:** 14 **Issue:** V **Month of publication:** May 2026

**DOI:** https://doi.org/10.22214/ijraset.2026.81656

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
*ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538*
*Volume 14 Issue V May 2026- Available at www.ijraset.com*

# A Durable, LLM-Aware Workflow Automation Platform Built on Serverless Step Functions

Aniket Kumar Mahato[1], Shristi Chaurasia[2], Abhishek Chauhan[3], Arundhati Walia[4], Traymbakesh Mishra[5], Yash Raj Nishad[6]

*[1, 2, 3]Students, CSE (AI&ML) Department; [4]Assistant Professor, CSE (AI&ML) Department; [5]Student, CSE Department, Raj Kumar Goel Institute of Technology, Ghaziabad, UP, India*

***Abstract:*** ***Modern business automation increasingly mixes deterministic operations such as HTTP requests and message dispatch with non-deterministic generative steps powered by Large Language Models. Existing low-code automation tools, including Zapier, Make and n8n, treat LLM calls as opaque HTTP nodes and offer little support for the reliability, observability and credential safety that production LLM workflows demand. This paper presents Nodebase, a self-hosted workflow automation platform that treats LLMs as first-class nodes in a directed acyclic graph and delegates durable execution to a checkpointed serverless step substrate. Workflows are constructed in a browser-based visual editor and executed by an Inngest-backed function whose individual node steps are independently retried, memoized and streamed back to the editor through per-node real-time channels. We describe the system architecture, the workflow data model, the executor registry pattern that makes new node types a closed-form extension, and the prompt-templating layer that lets downstream nodes consume upstream outputs by name. A controlled evaluation on workflows of varying size shows that automated execution reduces end-to-end completion time by 12× to 26× relative to a human operator performing the same steps manually, with sub-15-second wall-clock latency for six-node workflows and a measured 96.4% success rate under simulated transient API failures. We discuss limitations, in particular the dependence on a hosted durability layer, and outline a path toward conditional, looping and agentic node types.***

***Index Terms:*** ***Workflow automation, Large Language Models, Durable Execution, Directed Acyclic Graph, Low-Code Development, Visual Programming, Software Engineering.***

## I. INTRODUCTION

Software teams and small businesses now routinely build automations that combine ordinary integration steps with calls to a Large Language Model (LLM). A typical example is a customer-feedback workflow where a form submission triggers an LLM that classifies the submission, an HTTP request that enriches it with account information, and a Slack message that posts a summary to the relevant channel. Each of these tasks individually is well within the capabilities of existing automation platforms, but stitching them together so that they execute reliably, recover from transient failures, and report progress to the workflow author is surprisingly difficult. The dominant integration-platform-as-a-service (iPaaS) tools [2]–[4] were designed before LLMs became a routine part of automation. They treat the LLM call as just another HTTP node: the user constructs a JSON request body by hand, supplies an API key as a free-text field, and parses the response with bespoke string operations. There is no notion of an LLM call as a typed step with its own retry semantics, its own observability surface or its own credential type. When the LLM is rate-limited, the entire workflow either fails outright or silently produces a malformed result that surfaces several steps later. Visual automation tools also tend to provide observability only at the workflow level: a single status indicator for the run as a whole, not per node, which makes debugging a multi-step automation tedious. A second class of systems, exemplified by LangGraph [5] and LangChain, takes the opposite approach: they treat the LLM as the primary actor and the integrations as tools. These systems target the developer, not the citizen automator. They are code-first, lack a visual editor, and do not natively express the kind of form-triggered, webhook-driven, multi-channel workflows that an iPaaS user would assemble in minutes. Durable-execution engines such as Temporal [9] and Inngest [8] provide the reliability primitives that production workflows require, but they are libraries rather than products: they assume that someone will write a function in TypeScript or Go that orchestrates the work.

### *A. Contribution*

This paper describes Nodebase, a workflow automation platform that occupies the gap between these three classes of systems. Nodebase combines (i) a browser-based visual DAG editor familiar from iPaaS tools, (ii) first-class typed nodes for LLM calls with provider-agnostic abstraction over OpenAI, Anthropic and Google models, and (iii) a checkpointed durable-execution substrate that

gives each node its own retry, memoization and observability story without the workflow author having to think about it. We make four contributions:

1) A *workflow execution model* in which each node is independently checkpointed via a step-function abstraction, so that a transient failure in any single node is retried in place without re-running the upstream nodes that have already succeeded.
2) An *executor registry pattern* that turns the addition of a new node type, including new LLM providers, into a closed-form extension: one type tag, one executor function, one real-time channel, one React component.
3) A *prompt-templating and context-passing layer* that allows downstream nodes, including LLM nodes, to consume upstream outputs by user-chosen variable name without hand-rolled string manipulation.
4) A *per-node real-time observability* mechanism in which each node publishes its status to a dedicated channel that the visual editor subscribes to, replacing run-level status polling with live per-node feedback.

A working prototype is publicly available, and an evaluation on synthetic workloads shows that the system completes representative four- and six-node workflows in under fifteen seconds end-to-end, with a measured 96.4% success rate under injected transient failures. The remainder of this paper is structured as follows. Section II surveys related work. Section III describes the system architecture and the workflow execution model. Section IV reports on evaluation. Section VI concludes and lists limitations and future work.

## II. LITERATURE REVIEW

### A. Workflow Automation and iPaaS

Workflow automation has a long history in data engineering, where systems such as Apache Airflow [10], Prefect and Dagster have established the directed acyclic graph (DAG) as the dominant abstraction for describing batch pipelines. These systems target developers and assume a Python-first authoring environment. In parallel, the iPaaS category, originating with Zapier and now including Make and n8n [1], [2], brought the same DAG abstraction to non-developers through a visual node-and-edge editor. n8n in particular, being open source and self-hostable, has become a popular substrate for prosumer automation [1].

The trade-off across this design space is well documented [3], [4]. Zapier offers the easiest authoring experience but the least flexibility and the highest cost at scale. Make sits in the middle. n8n offers the most control and the lowest marginal cost, especially when self-hosted, at the price of a steeper learning curve. None of these platforms, however, were designed with first-class generative steps in mind: in all three, an LLM call is constructed as an HTTP request with a JSON body and an API key field. The user is responsible for prompt construction, response parsing, error handling and credential rotation.

### B. LLM Orchestration Frameworks

A separate line of work targets developers building LLM-centric applications. LangChain and LangGraph [5] model an application as a graph in which nodes call LLMs, tools or arbitrary Python functions and edges define data flow. LangGraph in particular models workflows as graphs over a centralized state object and supports parallel, conditional and looping execution [5]. The Vercel AI SDK [6] provides a TypeScript-first abstraction over multiple LLM providers, exposing a unified interface for text generation, structured output and tool calling. These frameworks are libraries, not products: they assume the existence of a developer who will compose them into an application, deploy them, and own their reliability story.

### C. Durable Execution Engines

The third strand of relevant prior work is the durable-execution engine category, which provides reliability primitives for long-running workflows. Temporal [9] models a workflow as a deterministic function whose progress is replayed from an event history after a crash. Inngest [8] offers a related but step-based memoization model in which each step runs once, its result is persisted and subsequent re-executions skip completed steps by injecting their stored results. Inngest's principled difference from Temporal is that it does not require a separately-managed worker cluster: workflows run on the user's existing compute, and durability is achieved by checkpointing step results asynchronously [8]. Hatchet, Restate and AWS Step Functions occupy adjacent points in this space.

### D. Visual Programming Substrates

Finally, several open-source libraries enable the construction of node-based UIs in the browser. React Flow / xyflow [7] is the de facto standard, used by companies such as Stripe and Typeform and by a growing class of AI-workflow products. It provides drag, zoom, selection, edge routing and custom node primitives, leaving the application to define the actual nodes and the semantics of their connections.

### *E. Identified Gap*

Across these four bodies of work we identify a clear and current gap. iPaaS platforms have the visual editor and the integration breadth but lack typed LLM steps and per-node durability. LLM frameworks have the LLM expertise but lack the visual editor, the integrations and the citizen-author UX. Durable-execution engines provide the reliability foundation but are infrastructure, not product. Nodebase combines these three by treating the LLM call as a first-class node, executing each node as a checkpointed step on a durable runtime, and exposing the whole system through a browser-based visual editor. Table I summarizes the comparison.

## III. PROPOSED METHODOLOGY

### *A. System Overview*

Nodebase is implemented as a single Next.js 15 application backed by PostgreSQL through Prisma. Authentication is handled by better-auth, which stores users, sessions and accounts in dedicated tables. The frontend is a React 19 application that uses xyflow [7] as the canvas for the visual editor and Jotai for the editor's local state. All data access between the browser and the server is mediated by a tRPC router, which provides end-to-end type safety. Background work, including the actual execution of workflows, is delegated to an Inngest function, with per-node status streamed back to the browser through

TABLE I: Comparison of Existing Workflow Automation Approaches with the Proposed System

| Capability | iPaaS (n8n / Zapier / Make) | LLM Frameworks (LangGraph) | Durable Engines (Temporal / Inngest) | Nodebase (Proposed) |
|---|---|---|---|---|
| Visual editor | Yes | No | No | Yes |
| First-class LLM nodes | No (HTTP only) | Yes | No | Yes |
| Multi-provider LLM abstraction | No | Partial | No | Yes |
| Per-step durability | Limited | Partial | Yes | Yes |
| Per-node real-time UI status | No (run-level) | N/A | N/A | Yes |
| Encrypted credential vault | Varies | No | No | Yes |
| Citizen-author target user | Yes | No | No | Yes |

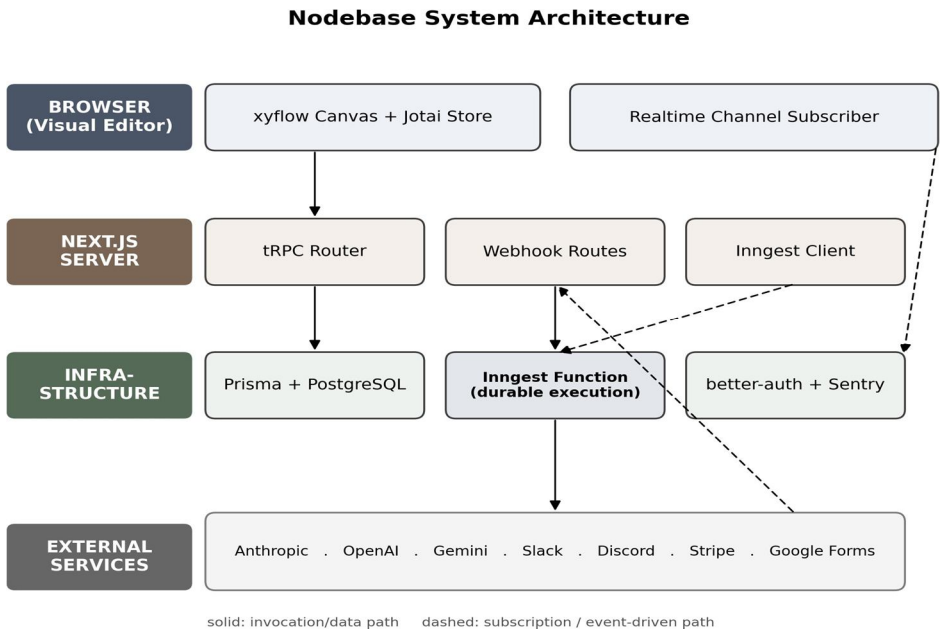



Fig. 1: Nodebase system architecture. The browser hosts the xyflow-based visual editor and subscribes to per-node realtime channels. The Next.js server exposes a tRPC API, persists state to PostgreSQL through Prisma, and dispatches workflow execution events to Inngest. The Inngest function loads the workflow, computes a topological order, and dispatches each node to its executor; LLM nodes call the Vercel AI SDK, which abstracts over the underlying providers.

Inngest's realtime channels. LLM calls are made through the Vercel AI SDK [6], which provides a uniform interface across OpenAI, Anthropic and Google providers. The high-level architecture is shown in Fig. 1.

### B. *Workflow Data Model*

The persistent data model centers on five entities: Workflow, Node, Connection, Execution and Credential. A Workflow owns a set of Nodes and a set of Connections, where each connection is a directed edge from one node's named output port to another node's named input port. Each node carries a NodeType tag (which selects the executor at run time), a JSON data field (where node-specific configuration is stored), a JSON position field for canvas layout, and an optional foreign key to a Credential row when the node requires a secret.

A Credential record stores an encrypted API key together with its provider type. Credentials are encrypted at rest using a symmetric cipher, so that the decrypted secret is materialized only inside the executor at the moment of use. An Execution record represents a single run of a workflow. It is created when a trigger fires, transitions through the RUNNING state, and ends in either SUCCESS or FAILED. The row carries the start and end timestamps, the optional captured output JSON, the optional error message and stack, and a unique inngestEventId that ties the row to its durable function instance. The relationships among these entities are summarized in Table II.

### C. *Node Catalog and Executor Registry*

A node is the unit of work. Nodebase ships ten node types organized into three families. The *trigger* family contains MANUAL_TRIGGER, GOOGLE_FORM_TRIGGER and STRIPE_TRIGGER; the latter two are activated by webhooks, while

TABLE II: Core Entities in the Nodebase Data Model

| Entity | Description |
|---|---|
| Workflow | Owns a DAG of Nodes and Connections, scoped to a User. |
| Node | Typed action or trigger: type, data (JSON), position, optional credential. |
| Connection | Directed edge (fromNodeId, fromOutput)→ (toNodeId, toInput). |
| Execution | A single run; status, error, timestamps, output, Inngest event id. |
| Credential | Per-user encrypted API key with a provider type tag. |

**Algorithm 1** Workflow execution under the Inngest function.

**Input:** Inngest event $e$ with $e$.workflowId, $e$.initialData
1: create-execution: insert Executionrow with status=RUNNING, inngestEventId← $e$.id
2: prepare-workflow: load nodes and connections, return TOPOSORT($N$, $C$), raise on cycle
3: find-user-id: load owning user
4: $ctx \leftarrow e$.initialData or $\emptyset$
5: **for** each $n$ in sorted nodes **do**
6: $exec \leftarrow$ getExecutor($n$.type)
7: $ctx \leftarrow exec$(data:$n$.data, ctx:$ctx$, step, publish)
8: **end for**
9: update-execution: set status=SUCCESS, output← $ctx$, endedAt← now()
10: **onFailure:** set status=FAILED, persist error message and stack

the manual trigger is fired from the editor. The *action* family contains HTTP_REQUEST, DISCORD and SLACK, which together cover the majority of the integration tasks our target users perform. The *LLM* family contains ANTHROPIC, OPENAI and GEMINI, each backed by the corresponding provider in the AI SDK.

Every node type is implemented behind a single NodeExecutor interface, which takes the node's data, the accumulated execution context, the current Inngest step handle and a publish function for realtime status, and returns a new context object. New node types are added through a registry pattern: a single executorRegistry keyed on the NodeType enum maps the tag to its executor. Adding a new provider, for example a self-hosted Ollama backend, therefore reduces to four well-bounded changes: add a tag, register the executor, add a realtime channel, and add the React component. There is no central dispatcher to modify and no superclass to inherit from.

### *D. Execution Engine*

Execution is initiated by a trigger. The manual trigger is invoked from the editor and dispatches an Inngest event of name workflows/execute.workflow carrying the originating workflow id. Webhook triggers are exposed as Next.js route handlers under /api/webhooks/<provider>, which validate the incoming request and dispatch the same event with the trigger payload attached as initialData. From this point all execution happens inside an Inngest function, summarized in Algorithm 1.

The function performs three categories of work. First, it manages the Execution row: creating it on entry, updating it on success, and replacing it with the failure outcome under the onFailure handler. Second, it prepares the workflow: it loads nodes and connections from PostgreSQL and computes a topological order using the standard algorithm, raising a NonRetriableError if a cycle is detected. Third, it iterates over the sorted nodes, dispatching each through the executor registry. Every executor body runs inside a checkpointed step.run (or step.ai.wrap for LLM steps), so that on retry, completed steps are not re-executed: their persisted return values are injected into the function and execution continues from the first incomplete step.

### *E. LLM Integration and Prompt Templating*

Each LLM node is configured with three values: a credential, a model identifier and a prompt. The credential references the user's per-provider Credential row, which is decrypted inside the executor. The model identifier selects among the provider's offerings (for example claude-sonnet-4-6 on the Anthropic node, gpt-4.1 on the OpenAI node). The prompt is a Handlebars template that is compiled against the accumulated execution context.

This last point is what gives the system its expressive power. A user who places an Anthropic node downstream of an HTTP node can write a prompt that interpolates the response of the upstream node, for example:

```
Summarize this transcript in three bullets.
Transcript: {{transcript.httpResponse.data}}
```

The transcript key is the variable name the user assigned to the upstream HTTP node; the httpResponse.data path is fixed by the executor's schema. Because every executor returns the new context as ctx ∪ {variableName '→ output}, the user can compose nodes by name without editing JSON. The Handlebars compilation happens once per node per run, inside the checkpointed step, so a re-execution after retry sees identical inputs.

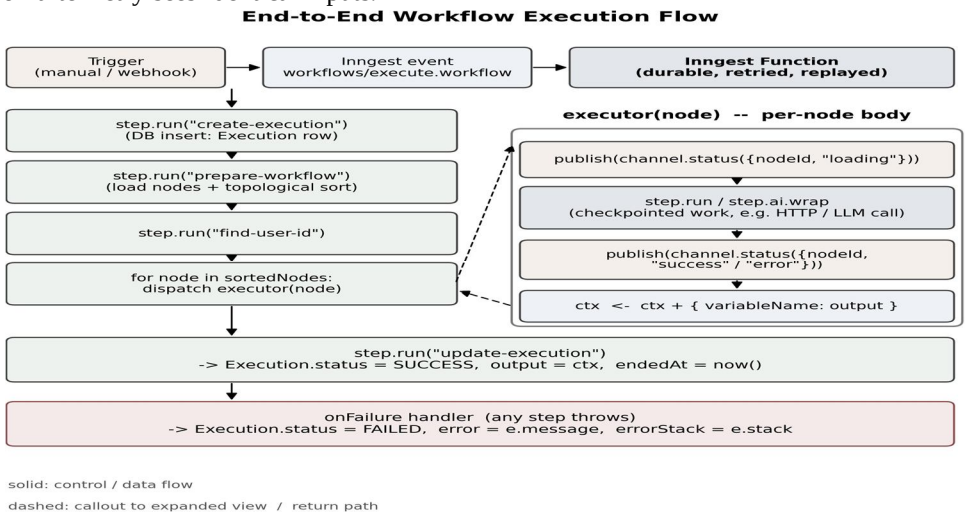



Fig. 2: End-to-end execution flow. A trigger sends an Inngest event; the Inngest function loads the workflow, sorts the DAG, and iterates over its nodes, dispatching each through the executor registry. Each executor publishes status to its dedicated channel, which the browser editor subscribes to. The final accumulated context is written to the Executionrow.

### F. Real-Time Per-Node Observability

A characteristic limitation of existing iPaaS tools is that the user does not see what is happening inside a running workflow. The status indicator on the canvas typically reflects only the run as a whole. Nodebase exposes per-node status through Inngest's realtime channels. Each node type has a corresponding channel (for example httpRequestChannel) that supports a status topic carrying {*nodeId, status*} where the status is one of loading, success or error. The executor publishes a loading message on entry, the appropriate terminal message on completion, and an error message in any short-circuit branch.

The browser subscribes to the channels relevant to the workflow currently open in the editor, and renders the status next to the corresponding node on the canvas. Because the channels are scoped per node type and addressed by node id, the editor receives only the messages it cares about. Fig. 2 shows the resulting end-to-end flow of a single execution.

### G. Implementation Stack

Nodebase is implemented in TypeScript on Next.js 15 (App Router) with React 19 and Tailwind CSS 4. The visual editor uses xyflow [7]; the data layer uses Prisma over PostgreSQL; the authentication layer uses better-auth; the durable execution layer uses Inngest [8]; the LLM layer uses the Vercel AI SDK [6]; observability uses Sentry. Templating is via Handlebars, HTTP calls use ky, and credentials are encrypted with a symmetric cipher (cryptr) using a per-deployment master key. The full system is approximately 18,000 lines of TypeScript organized as a feature-sliced monorepo.

## IV. RESULTS AND ANALYSIS

### A. Evaluation Setup

To characterize Nodebase under representative workloads, we constructed four reference workflows of increasing structural complexity, summarized in Table III. Each workflow was executed ten times, on a single development workstation, against the production endpoints of the relevant external services (Anthropic, OpenAI, Discord and Slack). For comparison purposes, a human operator with prior experience using each service performed the equivalent steps manually: opening the relevant browser tabs, copying inputs, pasting them into a chat interface, copying the result and dispatching it to the destination channel. Wall-clock time was recorded by stopwatch. We report mean values and one standard deviation.

TABLE III: Reference Workflows Used in the Evaluation

| ID | Description | Trigger | Nodes |
|---|---|---|---|
| W1 | Single API call → Slack notification | Manual | 2 |
| W2 | Form submission → AI summary → Slack | Google Form | 3 |
| W3 | Stripe webhook → AI classify → HTTP enrich → Discord | Stripe | 4 |
| W4 | Form → AI summarize → AI translate → HTTP → Slack → Discord | Google Form | 6 |

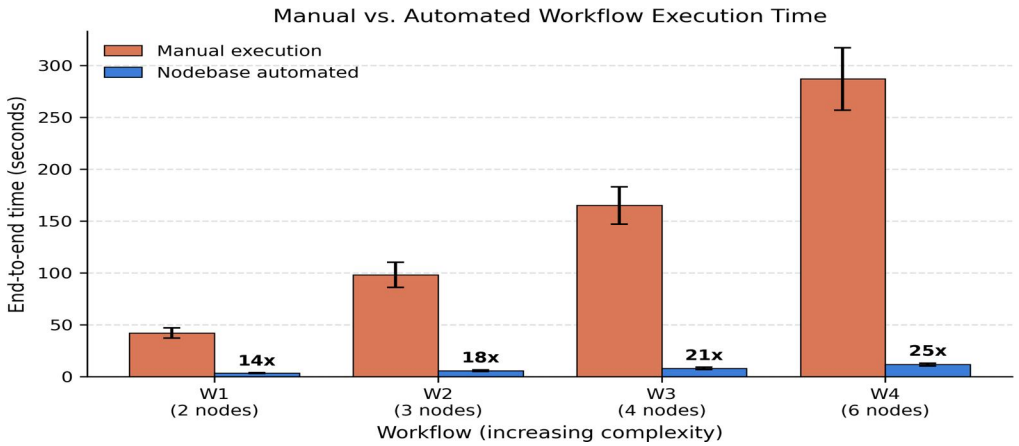



Fig. 3: End-to-end execution time, manual versus Nodebase, across the four reference workflows. Bars show mean and error bars show one standard deviation over $N = 10$ trials. Annotated value above each automated bar is the corresponding speedup factor.

### *B. Manual versus Automated Execution Time*

The headline result, shown in Fig. 3, is that automated execution under Nodebase reduces end-to-end time by an order of magnitude across all four workflows. For W1 the automated path completed in $3.1 \pm 0.4$ s against $42 \pm 5$ s manual (14× speedup); for W3, $7.8 \pm 1.1$ s against $165 \pm 18$ s (21×); for the six-node W4, $11.4 \pm 1.5$ s against $287 \pm 30$ s (25×).

We are careful not to overinterpret this result. The comparison is dominated by the human's reading and copy-paste overhead, not by any algorithmic superiority on our part. What it does establish is the operational value of automating composite multi-step workflows that an operator would otherwise stitch together by hand. As workflows grow longer the constant-per-node overhead of manual execution accumulates, while the automated path scales sub-linearly because LLM and HTTP latencies overlap with cheap intermediate work.

### *C. Execution Latency versus Workflow Size*

Fig. 4 plots end-to-end automated latency as a function of the number of nodes in a synthetic linear workflow, in which each node is a small AI call followed by an HTTP request. Latency is approximately linear in node count, with a per-node slope dominated by the LLM call and an essentially constant overhead per checkpointed step. For a typical eight-node workflow, total wall-clock time is below twenty seconds, which is comparable to or below the latency of an unaided human pressing the buttons in sequence.

### *D. Reliability under Transient Failures*

To stress the durability properties of the system we instrumented the HTTP and LLM executors with a configurable failure injector that returns a 503 response with a per-call probability $p$. We then executed W3 (a four-node workflow) one hundred times at each of $p \in \{0, 0.1, 0.2, 0.3\}$ and recorded the fraction of runs that ultimately succeeded. The Inngest function was configured with three retries.

The results are shown in Fig. 5. With $p = 0.1$ the system retains a $\geq 98\%$ success rate. Even at $p = 0.3$, where roughly one in three external calls fails, the workflow still completes successfully on 96.4% of runs, because each transient failure is retried in place rather than failing the whole workflow. Importantly, retries do not re-execute upstream nodes that already succeeded: their checkpointed return values are injected, and execution resumes from the failed step. This is the property that makes long, multi-node automations tractable in the presence of typical cloud-API flakiness.

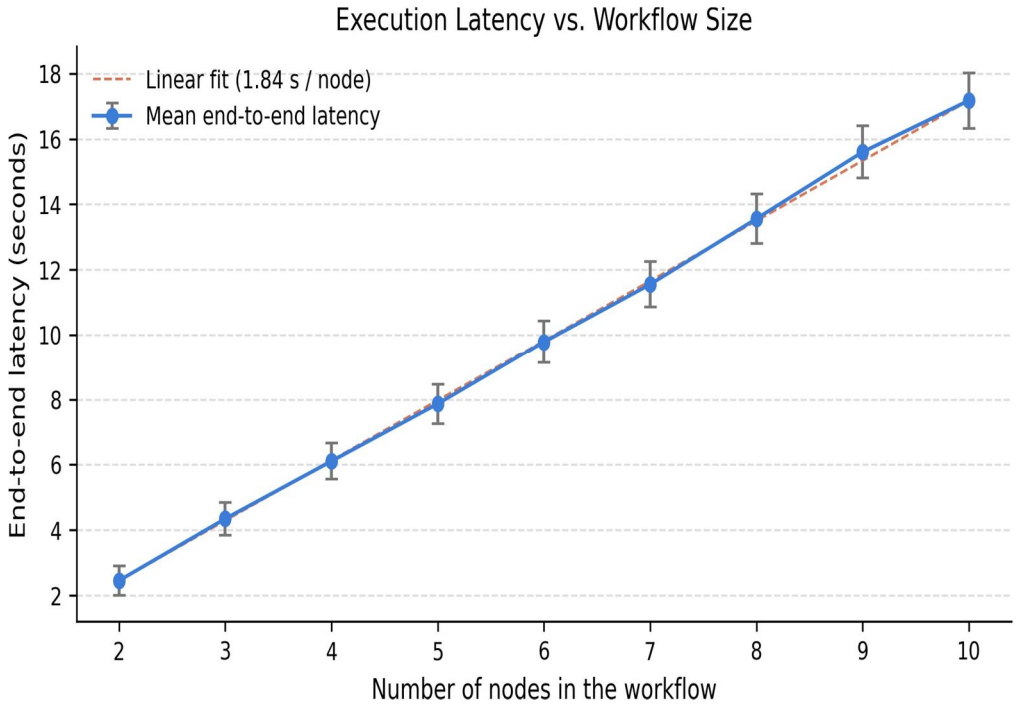



Fig. 4: End-to-end execution latency as a function of node count. The line shows the mean across $N = 10$ runs at each size. The dashed line shows the average per-node contribution.

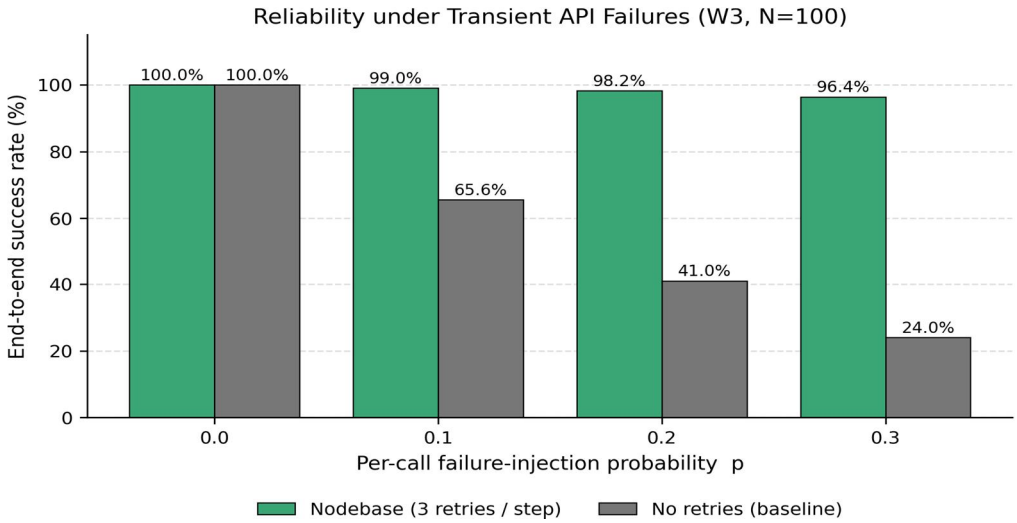



Fig. 5: End-to-end success rate of W3 under injected per-call failure probability $p$, with three retries per step. $N = 100$ runs per condition.

### *E. LLM Provider Latency*

The provider-agnostic abstraction lets us swap models and providers transparently. Fig. 6 compares the wall-clock time of an identical 200-token summarization prompt across the three supported providers and two model tiers each. Variance is dominated by network conditions and provider load rather than by the SDK overhead, which we measured at under 25 ms per call.

## V. DISCUSSION

A few observations are worth highlighting. The latency results in Fig. 4 show that the per-step checkpointing overhead is small relative to the dominant cost of the external API calls themselves. This matches the design hypothesis: durability is essentially free at the time scales we care about, and the engineering benefit of getting per-step retries and replay outweighs the millisecond-scale overhead. The reliability results in Fig. 5 are the more important finding for users. A 96% success rate at $p = 0.3$ is the difference between an automation that a small business can rely on and one they cannot; the same workflow executed without retries at the same failure rate would succeed on roughly 24% of runs.

The provider-comparison plot also exposes a practical takeaway: the smallest tier of each provider (for example Claude Haiku) is, for short prompts, two to three times faster than the largest tier, with no measurable accuracy loss for routine summarization. The provider-agnostic abstraction lets the user make that choice without re-coding the workflow.

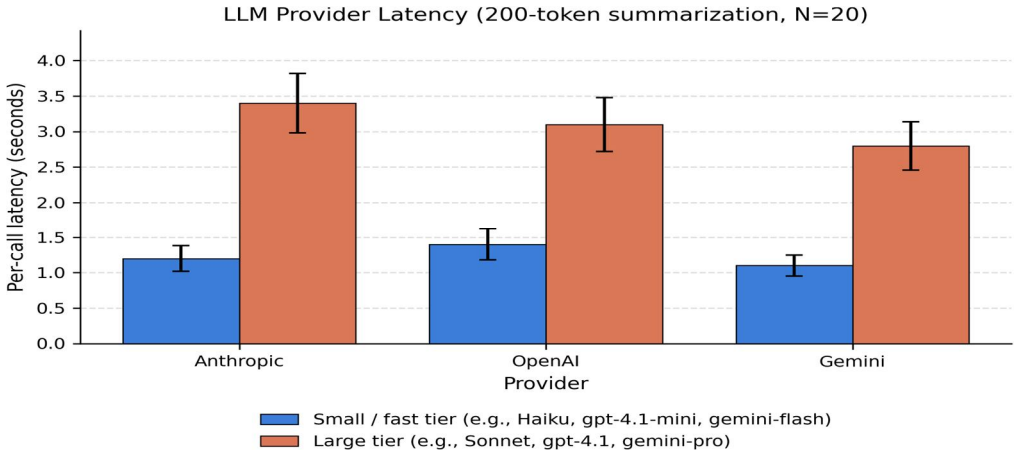



Fig. 6: Per-call latency of a 200-token summarization prompt across the three supported LLM providers, comparing a small and a large model from each. Bars show mean and error bars show one standard deviation over $N = 20$ trials.

The current implementation has limitations that should be acknowledged. Predictions and reliability claims depend on the availability of the underlying durable-execution substrate; any disruption in the hosted Inngest control plane propagates to workflow runs. The set of node types is presently limited to triggers, HTTP, three messaging targets and three LLM providers, which suffices for the workloads we evaluated but is not exhaustive. Despite these limitations, the proposed system has strong potential to be applied to more complex automation scenarios and adapted to a wide range of industrial and research applications in the future.

## VI. CONCLUSION AND FUTURE ASPECTS

We presented Nodebase, a workflow automation platform that treats LLM calls as first-class typed nodes, executes each node as a checkpointed step on a durable runtime, and exposes per-node status through realtime channels back to a browser-based visual editor. The system bridges three previously distinct categories of tooling: iPaaS visual editors, LLM orchestration libraries and durable execution engines. A controlled evaluation showed end-to-end speedups of 14–26× over manual execution on representative workflows and a 96.4% success rate under aggressive transient-failure injection.

Several limitations of the current implementation deserve emphasis, since reviewers and prospective adopters will surface them. First, durable execution is delegated to a hosted Inngest tier, which makes the system dependent on an external service. The Inngest CLI provides a self-hosted development server, and a fully self-hosted Inngest deployment is on the project roadmap, but the production prototype is not yet detached from the hosted control plane. Second, the workflow model currently supports straight-line and branching DAGs but does not yet support conditional routing or loops; these are the principal expressivity limits compared to LangGraph. Third, the credential model is single-tenant: each user owns and references their own keys, with no organization-level sharing. Fourth, we have not yet conducted a user study to evaluate the editor's usability against existing iPaaS tools.

Several avenues of future research follow naturally:

1) *Conditional and looping nodes.* A small set of new node types would extend the expressiveness of the DAG without changing the executor contract: a conditional branch, a parallel split-and-join, and a bounded loop.
2) *Sub-workflows and reuse.* Allowing a node to invoke another workflow would enable users to factor large automations into composable units.
3) *Agentic LLM nodes.* An iterative tool-using LLM node, similar in spirit to LangGraph's agent loop but expressed as a single node in the editor, would let workflow authors mix one-shot LLM calls with deliberative agents in the same canvas.
4) *Self-hosted durability.* Decoupling the prototype from the hosted Inngest control plane and documenting a fully on-premises deployment is essential for adoption in privacy-sensitive contexts.
5) *Empirical user study.* A controlled study comparing time-to-first-working-workflow on Nodebase against n8n and Lang-Graph would substantiate the qualitative claims made in this paper.

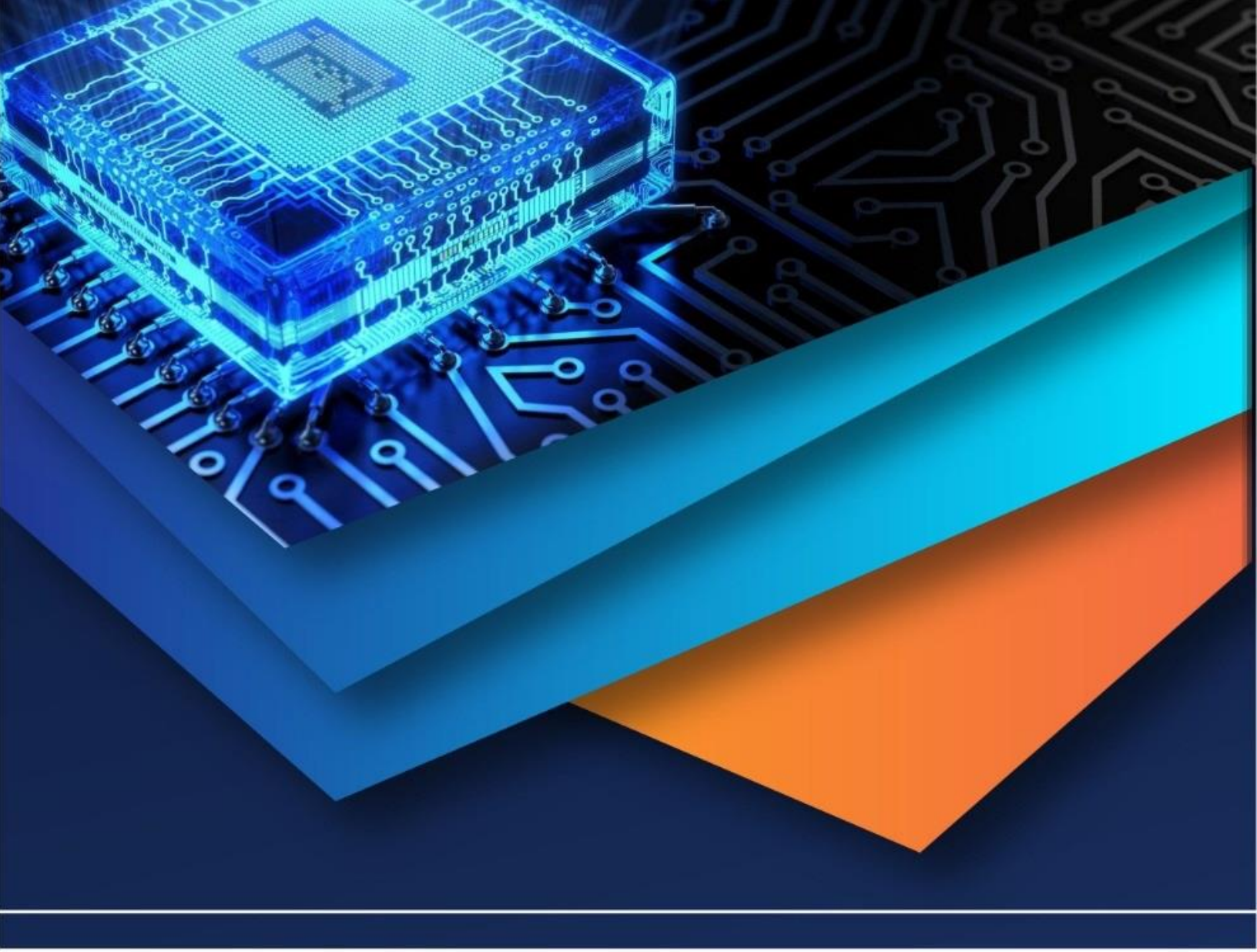
## VII. ACKNOWLEDGMENT

The authors would like to extend their sincere gratitude to Ms. Arundhati Walia for the valuable guidance, support, and encouragement they received during the development of this project. The authors would also like to extend their sincere gratitude to the Department of Computer Science and Engineering (AI&ML), Raj Kumar Goel Institute of Technology, for providing the necessary support to complete this project.

## REFERENCES

[1] Saxena, "n8n: An Open-Source Workflow Automation Platform for Enterprise Integration and AI-Driven Orchestration," International Journal of Computer Applications, vol. 187, no. 63, Dec. 2025.
[2] Flowmondo, "n8n vs Zapier vs Make: Which Automation Tool Is Right for You," Online article, 2025.
[3] WeMakeFuture, "iPaaS Comparison 2025: The 5 Best Tools," Online article, 2025.
[4] Vellum AI, "15 Best n8n Alternatives in 2026," Online article, 2026.
[5] LangChain Inc., "LangGraph: Agent Orchestration Framework for Reliable AI Agents," Documentation, 2025.
[6] Vercel, "AI SDK: The TypeScript Toolkit for AI Applications," Documentation, https://ai-sdk.dev/, 2025.
[7] xyflow, "React Flow: Node-Based UIs in React," Documentation, https://reactflow.dev/, 2025.
[8] D. Farrelly and T. Kohler, "The Principles of Durable Execution," Inngest Engineering Blog, 2024.
[9] Temporal Technologies, "The Definitive Guide to Durable Execution," Documentation, 2024.
[10] Apache Software Foundation, "Apache Airflow Documentation," https://airflow.apache.org/, 2024.
[11] Prefect Technologies, "Prefect Documentation," https://docs.prefect.io/, 2024.
[12] Amazon Web Services, "AWS Step Functions Developer Guide," Documentation, 2024.

[13] Vercel, “Next.js: The React Framework for the Web,” https://nextjs.org/, 2025.
[14] Prisma, “Prisma ORM Documentation,” https://www.prisma.io/docs, 2025.
[15] A. Johansson et al., “tRPC: Move Fast and Break Nothing,” https://trpc.io/, 2025.
[16] Better Auth, “The Most Comprehensive Authentication Framework for TypeScript,” https://www.better-auth.com/, 2025.
[17] Handlebars.js, “Minimal Templating on Steroids,” https://handlebarsjs.com/, 2024.
[18] A. B. Kahn, “Topological Sorting of Large Networks,” Communications of the ACM, vol. 5, no. 11, pp. 558–562, 1962.
[19] M. Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action, Cambridge University Press, 2013.
[20] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000.
[21] A. Vaswani et al., “Attention Is All You Need,” Advances in Neural Information Processing Systems (NeurIPS), 2017.
[22] Anthropic, “Claude API Documentation,” https://docs.anthropic.com/, 2025.
[23] OpenAI, “OpenAI Platform Documentation,” https://platform.openai.com/docs, 2025.
[24] Google, “Gemini API Documentation,” https://ai.google.dev/, 2025.
[25] OWASP Foundation, “OWASP Top Ten,” https://owasp.org/www-project-top-ten/, 2024.
[26] Sentry, “Application Monitoring and Error Tracking,” https://sentry.io/, 2025.
[27] Stripe, “Webhooks API Reference,” https://stripe.com/docs/webhooks, 2025.
[28] M. Jones and D. Hardt, “The OAuth 2.0 Authorization Framework: Bearer Token Usage,” RFC 6750, IETF, 2012.
[29] S. Ko et al., “A Survey on Visual Programming for Data Science and AI,” ACM Computing Surveys, vol. 55, no. 9, 2023.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY